## Spark性能调优

## 九项基本原则

Spark性能优化的第一步，就是要在开发Spark作业的过程中注意和应用一些性能优化的基本原则。开发调优，就是要让大家了解以下一些Spark基本开发原则，包括：RDD lineage设计、算子的合理使用、特殊操作的优化等。在开发过程中，时时刻刻都应该注意以上原则，并将这些原则根据具体的业务以及实际的应用场景，灵活地运用到自己的Spark作业中。

<http://spark.apache.org/docs/latest/tuning.html>

### 原则一：避免创建重复的RDD

通常来说，我们在开发一个Spark作业时，首先是基于某个数据源（比如Hive表或HDFS文件）创建一个初始的RDD；接着对这个RDD执行某个算子操作，然后得到下一个RDD；以此类推，循环往复，直到计算出最终我们需要的结果。在这个过程中，多个RDD会通过不同的算子操作（比如map、reduce等）串起来，这个“RDD串”，就是RDD lineage，也就是“RDD的血缘关系链”。

我们在开发过程中要注意：对于同一份数据，只应该创建一个RDD，不能创建多个RDD来代表同一份数据。

一些Spark初学者在刚开始开发Spark作业时，或者是有经验的工程师在开发RDD lineage极其冗长的Spark作业时，可能会忘了自己之前对于某一份数据已经创建过一个RDD了，从而导致对于同一份数据，创建了多个RDD。这就意味着，我们的Spark作业会进行多次重复计算来创建多个代表相同数据的RDD，进而增加了作业的性能开销。

* 一个简单的例子

*// 需要对名为“hello.txt”的HDFS文件进行一次map操作，再进行一次reduce操作。也就是说，需要对一份数据执行两次算子操作。*

*// 错误的做法：对于同一份数据执行多次算子操作时，创建多个RDD。// 这里执行了两次textFile方法，针对同一个HDFS文件，创建了两个RDD出来，然后分别对每个RDD都执行了一个算子操作。// 这种情况下，Spark需要从HDFS上两次加载hello.txt文件的内容，并创建两个单独的RDD；第二次加载HDFS文件以及创建RDD的性能开销，很明显是白白浪费掉的。*

val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")

rdd1.map(...)

val rdd2 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")

rdd2.reduce(...)

*// 正确的用法：对于一份数据执行多次算子操作时，只使用一个RDD。// 这种写法很明显比上一种写法要好多了，因为我们对于同一份数据只创建了一个RDD，然后对这一个RDD执行了多次算子操作。// 但是要注意到这里为止优化还没有结束，由于rdd1被执行了两次算子操作，第二次执行reduce操作的时候，还会再次从源头处重新计算一次rdd1的数据，因此还是会有重复计算的性能开销。// 要彻底解决这个问题，必须结合“原则三：对多次使用的RDD进行持久化”，才能保证一个RDD被多次使用时只被计算一次。*

val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")

rdd1.map(...)

rdd1.reduce(...)

### 原则二：尽可能复用同一个RDD

除了要避免在开发过程中对一份完全相同的数据创建多个RDD之外，在对不同的数据执行算子操作时还要尽可能地复用一个RDD。比如说，有一个RDD的数据格式是key-value类型的，另一个是单value类型的，这两个RDD的value数据是完全一样的。那么此时我们可以只使用key-value类型的那个RDD，因为其中已经包含了另一个的数据。对于类似这种多个RDD的数据有重叠或者包含的情况，我们应该尽量复用一个RDD，这样可以尽可能地减少RDD的数量，从而尽可能减少算子执行的次数。

* 一个简单的例子

*// 错误的做法。*

*// 有一个<Long, String>格式的RDD，即rdd1。// 接着由于业务需要，对rdd1执行了一个map操作，创建了一个rdd2，而rdd2中的数据仅仅是rdd1中的value值而已，也就是说，rdd2是rdd1的子集。*

JavaPairRDD<Long, String> rdd1 = ...

JavaRDD<String> rdd2 = rdd1.map(...)

*// 分别对rdd1和rdd2执行了不同的算子操作。*

rdd1.reduceByKey(...)

rdd2.map(...)

*// 正确的做法。*

*// 上面这个case中，其实rdd1和rdd2的区别无非就是数据格式不同而已，rdd2的数据完全就是rdd1的子集而已，却创建了两个rdd，并对两个rdd都执行了一次算子操作。// 此时会因为对rdd1执行map算子来创建rdd2，而多执行一次算子操作，进而增加性能开销。*

*// 其实在这种情况下完全可以复用同一个RDD。// 我们可以使用rdd1，既做reduceByKey操作，也做map操作。// 在进行第二个map操作时，只使用每个数据的tuple.\_2，也就是rdd1中的value值，即可。*

JavaPairRDD<Long, String> rdd1 = ...

rdd1.reduceByKey(...)

rdd1.map(tuple.\_2...)

*// 第二种方式相较于第一种方式而言，很明显减少了一次rdd2的计算开销。// 但是到这里为止，优化还没有结束，对rdd1我们还是执行了两次算子操作，rdd1实际上还是会被计算两次。// 因此还需要配合“原则三：对多次使用的RDD进行持久化”进行使用，才能保证一个RDD被多次使用时只被计算一次。*

### 原则三：对多次使用的RDD进行持久化

当你在Spark代码中多次对一个RDD做了算子操作后，恭喜，你已经实现Spark作业第一步的优化了，也就是尽可能复用RDD。此时就该在这个基础之上，进行第二步优化了，也就是要保证对一个RDD执行多次算子操作时，这个RDD本身仅仅被计算一次。

Spark中对于一个RDD执行多次算子的默认原理是这样的：每次你对一个RDD执行一个算子操作时，都会重新从源头处计算一遍，计算出那个RDD来，然后再对这个RDD执行你的算子操作。这种方式的性能是很差的。

因此对于这种情况，我们的建议是：对多次使用的RDD进行持久化。此时Spark就会根据你的持久化策略，将RDD中的数据保存到内存或者磁盘中。以后每次对这个RDD进行算子操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后执行算子，而不会从源头处重新计算一遍这个RDD，再执行算子操作。

* 对多次使用的RDD进行持久化的代码示例

*// 如果要对一个RDD进行持久化，只要对这个RDD调用cache()和persist()即可。*

*// 正确的做法。// cache()方法表示：使用非序列化的方式将RDD中的数据全部尝试持久化到内存中。// 此时再对rdd1执行两次算子操作时，只有在第一次执行map算子时，才会将这个rdd1从源头处计算一次。// 第二次执行reduce算子时，就会直接从内存中提取数据进行计算，不会重复计算一个rdd。*

val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").cache()

rdd1.map(...)

rdd1.reduce(...)

*// persist()方法表示：手动选择持久化级别，并使用指定的方式进行持久化。// 比如说，StorageLevel.MEMORY\_AND\_DISK\_SER表示，内存充足时优先持久化到内存中，内存不充足时持久化到磁盘文件中。// 而且其中的\_SER后缀表示，使用序列化的方式来保存RDD数据，此时RDD中的每个partition都会序列化成一个大的字节数组，然后再持久化到内存或磁盘中。// 序列化的方式可以减少持久化的数据对内存/磁盘的占用量，进而避免内存被持久化数据占用过多，从而发生频繁GC。*

val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").persist(StorageLevel.MEMORY\_AND\_DISK\_SER)

rdd1.map(...)

rdd1.reduce(...)

对于persist()方法而言，我们可以根据不同的业务场景选择不同的持久化级别。

* Spark的持久化级别

| **持久化级别** | **含义解释** |
| --- | --- |
| MEMORY\_ONLY | 使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会进行持久化。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是默认的持久化策略，使用cache()方法时，实际就是使用的这种持久化策略。 |
| MEMORY\_AND\_DISK | 使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。 |
| MEMORY\_ONLY\_SER | 基本含义同MEMORY\_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。 |
| MEMORY\_AND\_DISK\_SER | 基本含义同MEMORY\_AND\_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。 |
| DISK\_ONLY | 使用未序列化的Java对象格式，将数据全部写入磁盘文件中。 |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, 等等. | 对于上述任意一种持久化策略，如果加上后缀\_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了。 |

* 如何选择一种最合适的持久化策略

默认情况下，性能最高的当然是MEMORY\_ONLY，但前提是你的内存必须足够足够大，可以绰绰有余地存放下整个RDD的所有数据。因为不进行序列化与反序列化操作，就避免了这部分的性能开销；对这个RDD的后续算子操作，都是基于纯内存中的数据的操作，不需要从磁盘文件中读取数据，性能也很高；而且不需要复制一份数据副本，并远程传送到其他节点上。但是这里必须要注意的是，在实际的生产环境中，恐怕能够直接用这种策略的场景还是有限的，如果RDD中数据比较多时（比如几十亿），直接用这种持久化级别，会导致JVM的OOM内存溢出异常。

如果使用MEMORY\_ONLY级别时发生了内存溢出，那么建议尝试使用MEMORY\_ONLY\_SER级别。该级别会将RDD数据序列化后再保存在内存中，此时每个partition仅仅是一个字节数组而已，大大减少了对象数量，并降低了内存占用。这种级别比MEMORY\_ONLY多出来的性能开销，主要就是序列化与反序列化的开销。但是后续算子可以基于纯内存进行操作，因此性能总体还是比较高的。此外，可能发生的问题同上，如果RDD中的数据量过多的话，还是可能会导致OOM内存溢出的异常。

如果纯内存的级别都无法使用，那么建议使用MEMORY\_AND\_DISK\_SER策略，而不是MEMORY\_AND\_DISK策略。因为既然到了这一步，就说明RDD的数据量很大，内存无法完全放下。序列化后的数据比较少，可以节省内存和磁盘的空间开销。同时该策略会优先尽量尝试将数据缓存在内存中，内存缓存不下才会写入磁盘。

通常不建议使用DISK\_ONLY和后缀为\_2的级别：因为完全基于磁盘文件进行数据的读写，会导致性能急剧降低，有时还不如重新计算一次所有RDD。后缀为\_2的级别，必须将所有数据都复制一份副本，并发送到其他节点上，数据复制以及网络传输会导致较大的性能开销，除非是要求作业的高可用性，否则不建议使用。

### 原则四：尽量避免使用shuffle类算子

如果有可能的话，要尽量避免使用shuffle类算子。因为Spark作业运行过程中，最消耗性能的地方就是shuffle过程。shuffle过程，简单来说，就是将分布在集群中多个节点上的同一个key，拉取到同一个节点上，进行聚合或join等操作。比如reduceByKey、join等算子，都会触发shuffle操作。

shuffle过程中，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。因此在shuffle过程中，可能会发生大量的磁盘文件读写的IO操作，以及数据的网络传输操作。磁盘IO和网络数据传输也是shuffle性能较差的主要原因。

因此在我们的开发过程中，能避免则尽可能避免使用reduceByKey、join、distinct、repartition等会进行shuffle的算子，尽量使用map类的非shuffle算子。这样的话，没有shuffle操作或者仅有较少shuffle操作的Spark作业，可以大大减少性能开销。

* Broadcast与map进行join代码示例

*// 传统的join操作会导致shuffle操作。*

*// 因为两个RDD中，相同的key都需要通过网络拉取到一个节点上，由一个task进行join操作。*

**val** rdd3 = rdd1.join(rdd2)

*// Broadcast+map的join操作，不会导致shuffle操作。*

*// 使用Broadcast将一个数据量较小的RDD作为广播变量。*

**val** rdd2Data = rdd2.collect()

**val** rdd2DataBroadcast = sc.broadcast(rdd2Data)

*// 在rdd1.map算子中，可以从rdd2DataBroadcast中，获取rdd2的所有数据。// 然后进行遍历，如果发现rdd2中某条数据的key与rdd1的当前数据的key是相同的，那么就判定可以进行join。// 此时就可以根据自己需要的方式，将rdd1当前数据与rdd2中可以连接的数据，拼接在一起（String或Tuple）。*

**val** rdd3 = rdd1.map(rdd2DataBroadcast...)

*// 注意，以上操作，建议仅仅在rdd2的数据量比较少（比如几百M，或者一两G）的情况下使用。// 因为每个Executor的内存中，都会驻留一份rdd2的全量数据。*

### 原则五：使用map-side预聚合的shuffle操作

优先使用reduceByKey替换groupByKey+聚合，reduceByKey有预聚合

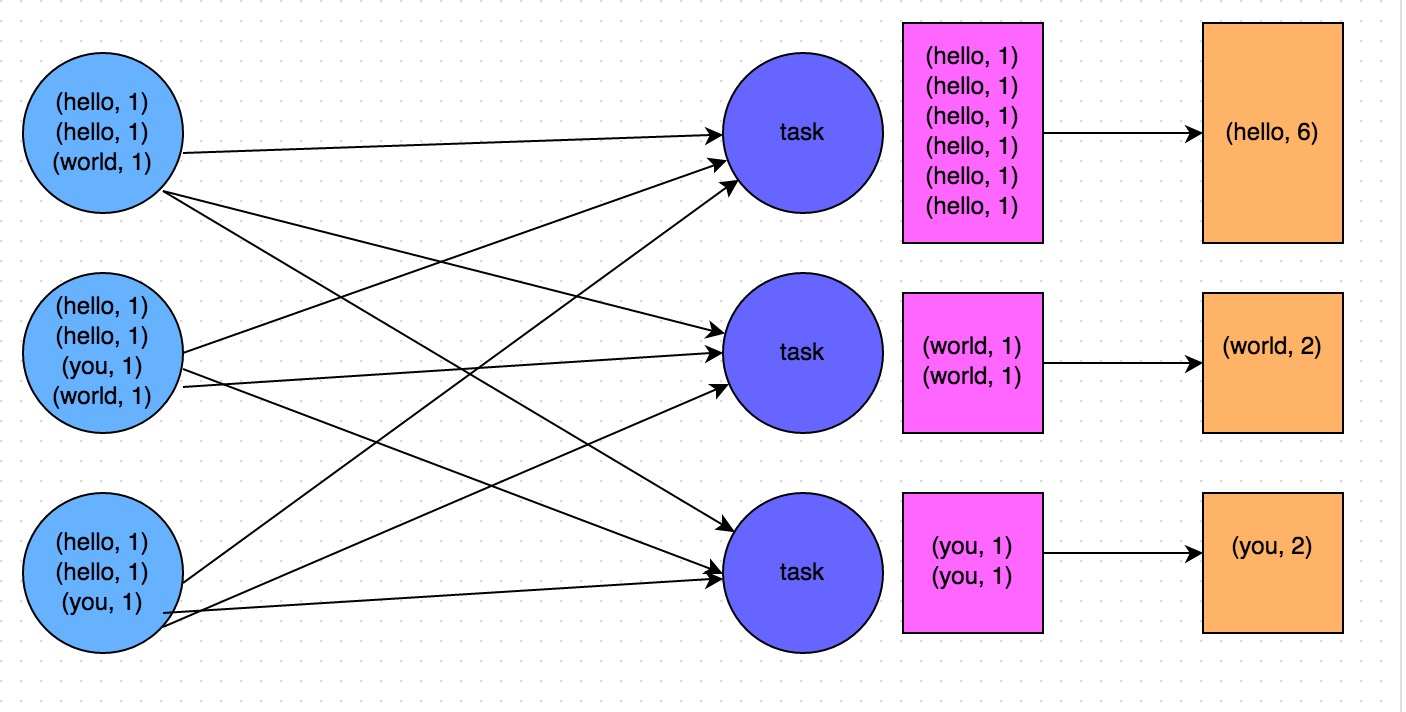
如果因为业务需要，一定要使用shuffle操作，无法用map类的算子来替代，那么尽量使用可以map-side预聚合的算子。

所谓的map-side预聚合，说的是在每个节点本地对相同的key进行一次聚合操作，类似于MapReduce中的本地combiner。map-side预聚合之后，每个节点本地就只会有一条相同的key，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘IO以及网络传输开销。

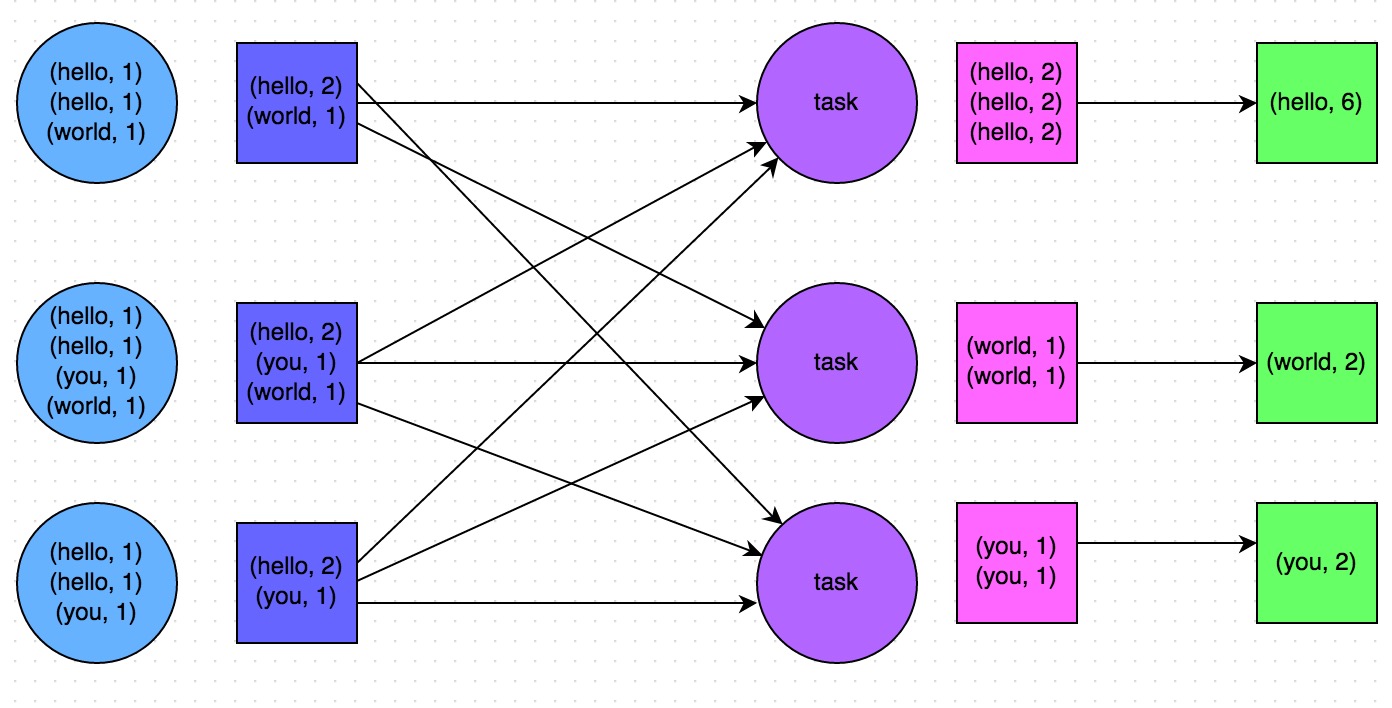
通常来说，在可能的情况下，建议使用reduceByKey或者aggregateByKey算子来替代掉groupByKey算子。因为reduceByKey和aggregateByKey算子都会使用用户自定义的函数对每个节点本地的相同key进行预聚合。而groupByKey算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

比如如下两幅图，就是典型的例子，分别基于reduceByKey和groupByKey进行单词计数。其中

* groupByKey的原理图，可以看到，没有进行任何本地聚合时，所有数据都会在集群节点之间传输；



* reduceByKey的原理图，可以看到，每个节点本地的相同key数据，都进行了预聚合，然后才传输到其他节点上进行全局聚合。



### 原则六：使用高性能的算子

除了shuffle相关的算子有优化原则之外，其他的算子也都有着相应的优化原则。

* 使用reduceByKey/aggregateByKey替代groupByKey

详情见“原则五：使用map-side预聚合的shuffle操作”。

* 使用mapPartitions替代普通map

mapPartitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。但是有的时候，使用mapPartitions会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！

* 使用foreachPartitions替代foreach

原理类似于“使用mapPartitions替代map”，也是一次函数调用处理一个partition的所有数据，而不是一次函数调用处理一条数据。在实践中发现，foreachPartitions类的算子，对性能的提升还是很有帮助的。比如在foreach函数中，将RDD中所有数据写MySQL，那么如果是普通的foreach算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个数据库连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用foreachPartitions算子一次性处理一个partition的数据，那么对于每个partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。实践中发现，对于1万条左右的数据量写MySQL，性能可以提升30%以上。

* 使用filter之后进行coalesce操作

Filter会过滤数据，数据会减少，没必要使用太多分区，通过coalesce减少分区

通常对一个RDD执行filter算子过滤掉RDD中较多数据后（比如30%以上的数据），建议使用coalesce算子，手动减少RDD的partition数量，将RDD中的数据压缩到更少的partition中去。因为filter之后，RDD的每个partition中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个task处理的partition中的数据量并不是很多，有一点资源浪费，而且此时处理的task越多，可能速度反而越慢。因此用coalesce减少partition数量，将RDD中的数据压缩到更少的partition之后，只要使用更少的task即可处理完所有的partition。在某些场景下，对于性能的提升会有一定的帮助。

* 使用repartitionAndSortWithinPartitions替代repartition与sort类操作

repartitionAndSortWithinPartitions是Spark官网推荐的一个算子，官方建议，如果需要在repartition重分区之后，还要进行排序，建议直接使用repartitionAndSortWithinPartitions算子。因为该算子可以一边进行重分区的shuffle操作，一边进行排序。shuffle与sort两个操作同时进行，比先shuffle再sort来说，性能可能是要高的。

### 原则七：广播大变量

公共变量进行广播，如果不广播会将变量发送到各个task，如果通过广播会发送到每个worknode的内存缓存，次数广播的变量数据偏大，如果数据很少也就没必要广播

有时在开发过程中，会遇到需要在算子函数中使用外部变量的场景（尤其是大变量，比如100M以上的大集合），那么此时就应该使用Spark的广播（Broadcast）功能来提升性能。

在算子函数中使用到外部变量时，默认情况下，Spark会将该变量复制多个副本，通过网络传输到task中，此时每个task都有一个变量副本。如果变量本身比较大的话（比如100M，甚至1G），那么大量的变量副本在网络中传输的性能开销，以及在各个节点的Executor中占用过多内存导致的频繁GC，都会极大地影响性能。

因此对于上述情况，如果使用的外部变量比较大，建议使用Spark的广播功能，对该变量进行广播。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，而Executor中的task执行时共享该Executor中的那份变量副本。这样的话，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对Executor内存的占用开销，降低GC的频率。

* 广播大变量的代码示例

*// 以下代码在算子函数中，使用了外部的变量。// 此时没有做任何特殊操作，每个task都会有一份list1的副本。***val** list1 = ...

rdd1.map(list1...)

*// 以下代码将list1封装成了Broadcast类型的广播变量。// 在算子函数中，使用广播变量时，首先会判断当前task所在Executor内存中，是否有变量副本。// 如果有则直接使用；如果没有则从Driver或者其他Executor节点上远程拉取一份放到本地Executor内存中。// 每个Executor内存中，就只会驻留一份广播变量副本。***val** list1 = ...**val** list1Broadcast = sc.broadcast(list1)

rdd1.map(list1Broadcast...)

### 原则八：使用Kryo优化序列化性能

在Spark中，主要有三个地方涉及到了序列化：

\* 在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输（见“原则七：广播大变量”中的讲解）。

\* 将自定义的类型作为RDD的泛型类型时（比如JavaRDD，Student是自定义类型），所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现Serializable接口。

\* 使用可序列化的持久化策略时（比如MEMORY\_ONLY\_SER），Spark会将RDD中的每个partition都序列化成一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库，来优化序列化和反序列化的性能。Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介绍，Kryo序列化机制比Java序列化机制，性能高10倍左右。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说，这种方式比较麻烦。但从Spark 2.0.0版本开始，简单类型、简单类型数组、字符串类型的Shuffling RDDs 已经默认使用Kryo序列化方式了。

以下是使用Kryo的代码示例，我们只要设置序列化类，再注册要序列化的自定义类型即可（比如算子函数中使用到的外部变量类型、作为RDD泛型类型的自定义类型等）：

*public class MyClass* **implements** *KryoRegistrator{*

*@Override*

*public void registerClasses(Kryo kryo){*

*kryo.register(StartupReportLogs.class);*

*}*

*}*

*// 创建SparkConf对象。*

val conf = **new** SparkConf().setMaster(...).setAppName(...)

*// 设置序列化器为KryoSerializer。*

conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

*// 注册要序列化的自定义类型。*

conf.registerKryoClasses(Array(classOf[MyClass], classOf[MyClass2]))

### 原则九：优化数据结构

Java中，有三种类型比较耗费内存：

\* 对象，每个Java对象都有对象头、引用等额外的信息，因此比较占用内存空间。

\* 字符串，每个字符串内部都有一个字符数组以及长度等额外信息。

\* 集合类型，比如HashMap、LinkedList等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如Map.Entry。

因此Spark官方建议，在Spark编码实现中，特别是对于算子函数中的代码，尽量不要使用上述三种数据结构，尽量使用字符串替代对象，使用原始类型（比如Int、Long）替代字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低GC频率，提升性能。

但是在编码实践中发现，要做到该原则其实并不容易。因为我们同时要考虑到代码的可维护性，如果一个代码中，完全没有任何面向对象的抽象，那么对于后续的代码维护和修改，无疑是一场巨大的灾难。同理，如果所有操作都基于数组实现，而不使用HashMap、LinkedList等集合类型，那么对于我们的编码难度以及代码可维护性，也是一个极大的挑战。因此，在可能以及合适的情况下，使用占用内存较少的数据结构，但是前提是要保证代码的可维护性。

## N多配置参数

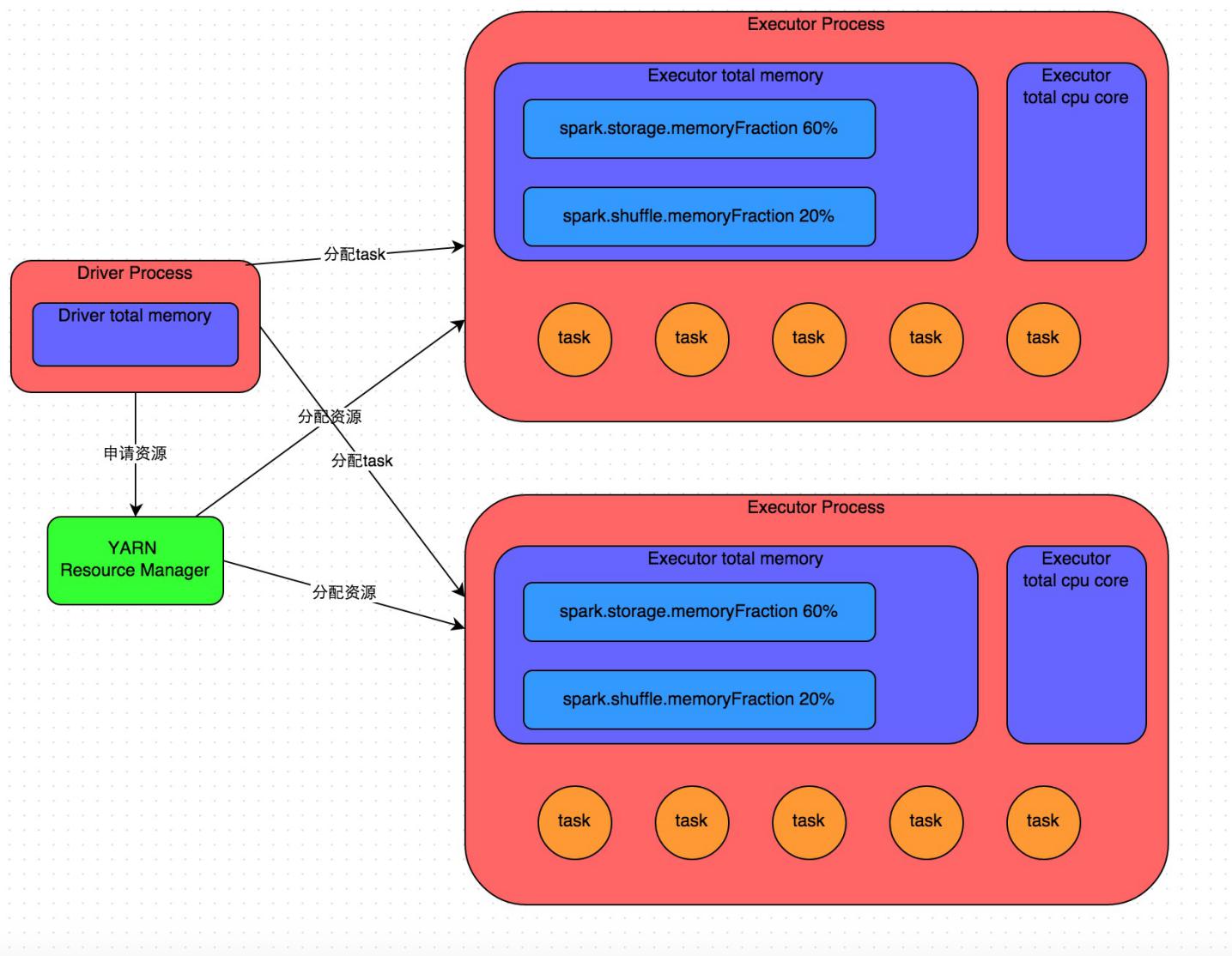
在开发完Spark作业之后，就该为作业配置合适的资源了。Spark的资源参数，基本都可以在spark-submit命令中作为参数设置。很多Spark初学者，通常不知道该设置哪些必要的参数，以及如何设置这些参数，最后就只能胡乱设置，甚至压根儿不设置。

资源参数设置的不合理，可能会导致没有充分利用集群资源，作业运行会极其缓慢；或者设置的资源过大，队列没有足够的资源来提供，进而导致各种异常。总之，无论是哪种情况，都会导致Spark作业的运行效率低下，甚至根本无法运行。

因此我们必须对Spark作业的资源使用原理有一个清晰的认识，并知道在Spark作业运行过程中，有哪些资源参数是可以设置的，以及如何设置合适的参数值。

<http://spark.apache.org/docs/latest/configuration.html>

### Spark作业基本运行原理



详细原理见上图。我们使用spark-submit提交一个Spark作业之后，这个作业就会启动一个对应的Driver进程。根据你使用的部署模式（deploy-mode）不同，Driver进程可能在本地启动，也可能在集群中某个工作节点上启动。Driver进程本身会根据我们设置的参数，占有一定数量的内存和CPU core。而Driver进程要做的第一件事情，就是向集群管理器（可以是Spark Standalone集群，也可以是其他的资源管理集群，国内使用较多的是YARN作为资源管理集群）申请运行Spark作业需要使用的资源，这里的资源指的就是Executor进程。YARN集群管理器会根据我们为Spark作业设置的资源参数，在各个工作节点上，启动一定数量的Executor进程，每个Executor进程都占有一定数量的内存和CPU core。

在申请到了作业执行所需的资源之后，Driver进程就会开始调度和执行我们编写的作业代码了。Driver进程会将我们编写的Spark作业代码分拆为多个stage，每个stage执行一部分代码片段，并为每个stage创建一批task，然后将这些task分配到各个Executor进程中执行。task是最小的计算单元，负责执行一模一样的计算逻辑（也就是我们自己编写的某个代码片段），只是每个task处理的数据不同而已。一个stage的所有task都执行完毕之后，会在各个节点本地的磁盘文件中写入计算中间结果，然后Driver就会调度运行下一个stage。下一个stage的task的输入数据就是上一个stage输出的中间结果。如此循环往复，直到将我们自己编写的代码逻辑全部执行完，并且计算完所有的数据，得到我们想要的结果为止。

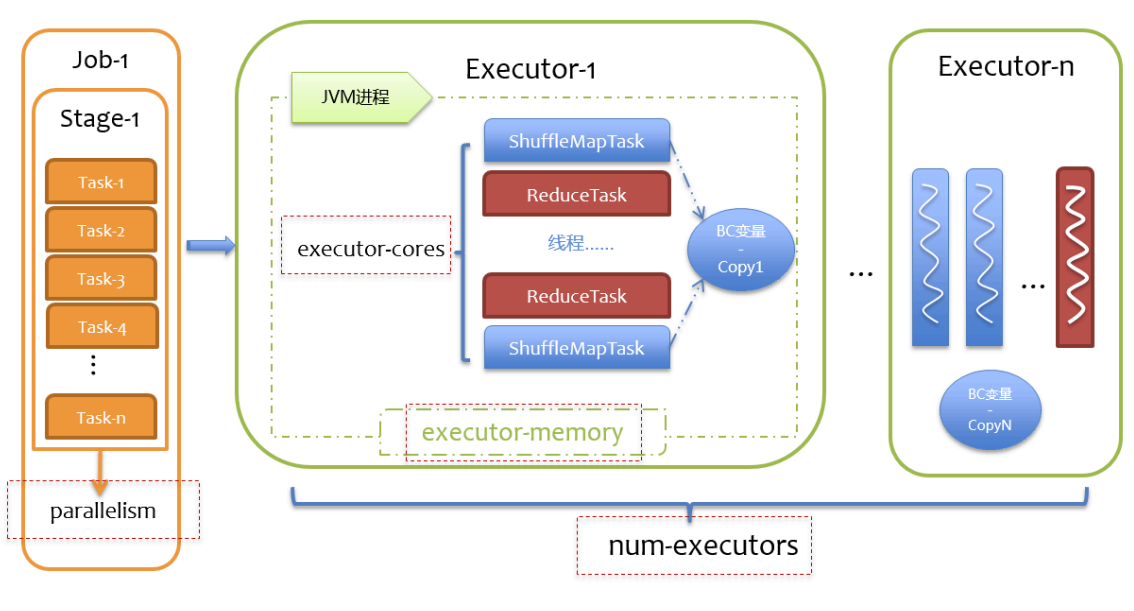
Spark是根据shuffle类算子来进行stage的划分。如果我们的代码中执行了某个shuffle类算子（比如reduceByKey、join等），那么就会在该算子处，划分出一个stage界限来。可以大致理解为，shuffle算子执行之前的代码会被划分为一个stage，shuffle算子执行以及之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个task可能都会从上一个stage的task所在的节点，去通过网络传输拉取需要自己处理的所有key，然后对拉取到的所有相同的key使用我们自己编写的算子函数执行聚合操作（比如reduceByKey()算子接收的函数）。这个过程就是shuffle。

当我们在代码中执行了cache/persist等持久化操作时，根据我们选择的持久化级别的不同，每个task计算出来的数据也会保存到Executor进程的内存或者所在节点的磁盘文件中。

因此Executor的内存主要分为三块：第一块是让task执行我们自己编写的代码时使用，默认是占Executor总内存的20%；第二块是让task通过shuffle过程拉取了上一个stage的task的输出后，进行聚合等操作时使用，默认也是占Executor总内存的20%；第三块是让RDD持久化时使用，默认占Executor总内存的60%。

task的执行速度是跟每个Executor进程的CPU core数量有直接关系的。一个CPU core同一时间只能执行一个线程。而每个Executor进程上分配到的多个task，都是以每个task一条线程的方式，多线程并发运行的。如果CPU core数量比较充足，而且分配到的task数量比较合理，那么通常来说，可以比较快速和高效地执行完这些task线程。

以上就是Spark作业的基本运行原理的说明，大家可以结合图片来理解。理解作业基本原理，是我们进行资源参数调优的基本前提。



### 提交参数

#### num-executors

参数说明：该参数用于设置Spark作业总共要用多少个Executor进程来执行。Driver在向YARN集群管理器申请资源时，YARN集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的Executor进程。这个参数非常之重要，如果不设置的话，默认只会给你启动少量的Executor进程，此时你的Spark作业的运行速度是非常慢的。

参数调优建议：每个Spark作业的运行一般设置50~100个左右的Executor进程比较合适，设置太少或太多的Executor进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。

#### executor-memory

参数说明：该参数用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能，而且跟常见的JVM OOM异常，也有直接的关联。

参数调优建议：每个Executor进程的内存设置4G~8G较为合适。但是这只是一个参考值，具体的设置还是得根据不同部门的资源队列来定。可以看看自己团队的资源队列的最大内存限制是多少，num-executors乘以executor-memory，是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的内存量最好不要超过资源队列最大总内存的1/3~1/2，避免你自己的Spark作业占用了队列所有的资源，导致别的同学的作业无法运行。

#### executor-cores

参数说明：该参数用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core数量越多，越能够快速地执行完分配给自己的所有task线程。

参数调优建议：Executor的CPU core数量设置为2~4个较为合适。同样得根据不同部门的资源队列来定，可以看看自己的资源队列的最大CPU core限制是多少，再依据设置的Executor数量，来决定每个Executor进程可以分配到几个CPU core。同样建议，如果是跟他人共享这个队列，那么num-executors \* executor-cores不要超过队列总CPU core的1/3~1/2左右比较合适，也是避免影响其他的作业运行。

#### driver-memory

参数说明：该参数用于设置Driver进程的内存。

参数调优建议：Driver的内存通常来说不设置，或者设置1G左右应该就够了。唯一需要注意的一点是，如果需要使用collect、take、top等算子将RDD的数据全部拉取到Driver上进行处理，那么必须确保Driver的内存足够大，否则会出现OOM内存溢出的问题。

### 常用conf参数

#### spark.default.parallelism

参数说明：该参数用于设置每个stage的默认task数量。这个参数极为重要，如果不设置可能会直接影响你的Spark作业性能。

参数调优建议：Spark作业的默认task数量为500~1000个较为合适。很多同学常犯的一个错误就是不去设置这个参数，那么此时就会导致Spark自己根据底层HDFS的block数量来设置task的数量，默认是一个HDFS block对应一个task。通常来说，Spark默认设置的数量是偏少的（比如就几十个task），如果task数量偏少的话，就会导致你前面设置好的Executor的参数都前功尽弃。试想一下，无论你的Executor进程有多少个，内存和CPU有多大，但是task只有1个或者10个，那么90%的Executor进程可能根本就没有task执行，也就是白白浪费了资源！因此Spark官网建议的设置原则是，设置该参数为num-executors \* executor-cores的2~3倍较为合适，比如Executor的总CPU core数量为300个，那么设置1000个task是可以的，此时可以充分地利用Spark集群的资源。

#### spark.storage.memoryFraction

参数说明：该参数用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6。也就是说，默认Executor 60%的内存，可以用来保存持久化的RDD数据。根据你选择的不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。

参数调优建议：如果Spark作业中，有较多的RDD持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降低了性能。但是如果Spark作业中的shuffle类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的gc导致运行缓慢（通过spark web ui可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

#### spark.shuffle.memoryFraction

例如：ruduceByKey，shuffle之后的reduce操作

参数说明：该参数用于设置shuffle过程中一个task拉取到上个stage的task的输出后，进行聚合操作时能够使用的Executor内存的比例，默认是0.2。也就是说，Executor默认只有20%的内存用来进行该操作。shuffle操作在进行聚合时，如果发现使用的内存超出了这个20%的限制，那么多余的数据就会溢写到磁盘文件中去，此时就会极大地降低性能。

参数调优建议：如果Spark作业中的RDD持久化操作较少，shuffle操作较多时，建议降低持久化操作的内存占比，提高shuffle操作的内存占比比例，避免shuffle过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此外，如果发现作业由于频繁的gc导致运行缓慢，意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

#### spark.sql.codegen

默认值为false，当它设置为true时，Spark SQL会把每条查询的语句在运行时编译为java的二进制代码。这有什么作用呢？它可以提高大型查询的性能，但是如果进行小规模的查询的时候反而会变慢，就是说直接用查询反而比将它编译成为java的二进制代码快。所以在优化这个选项的时候要视情况而定。

这个选项可以让Spark SQL把每条查询语句在运行前编译为java二进制代码，由于生成了专门运行指定查询的代码，codegen可以让大型查询或者频繁重复的查询明显变快，然而在运行特别快（1-2秒）的即时查询语句时，codegen就可能增加额外的开销（将查询语句编译为java二进制文件）。codegen还是一个实验性的功能，但是在大型的或者重复运行的查询中使用codegen

#### spark.sql.inMemoryColumnStorage.compressed

默认值为false 它的作用是自动对内存中的列式存储进行压缩

#### spark.sql.inMemoryColumnStorage.batchSize

默认值为1000 这个参数代表的是列式缓存时的每个批处理的大小。如果将这个值调大可能会导致内存不够的异常，所以在设置这个的参数的时候得注意你的内存大小

在缓存SchemaRDD（Row RDD）时，Spark SQL会安照这个选项设定的大小（默认为1000）把记录分组，然后分批次压缩。

太小的批处理会导致压缩比过低，而太大的话，比如当每个批处理的数据超过内存所能容纳的大小时，也有可能引发问题。

如果你表中的记录比价大（包含数百个字段或者包含像网页这样非常大的字符串字段），就可能需要调低批处理的大小来避免内存不够（OOM）的错误。如果不是在这样的场景下，默认的批处理 的大小是比较合适的，因为压缩超过1000条压缩记录时也基本无法获得更高的压缩比了。

#### spark.sql.parquet.compressed.codec

默认值为snappy 这个参数代表使用哪种压缩编码器。可选的选项包括uncompressed/snappy/gzip/lzo        uncompressed这个顾名思义就是不用压缩的意思

#### spark.speculation

推测执行优化机制采用了典型的以空间换时间的优化策略，它同时启动多个相同task（备份任务）处理相同的数据块，哪个完成的早，则采用哪个task的结果，这样可防止拖后腿Task任务出现，进而提高作业计算速度，但是，这样却会占用更多的资源，在集群资源紧缺的情况下，设计合理的推测执行机制可在多用少量资源情况下，减少大作业的计算时间。

检查逻辑代码中注释很明白，当成功的Task数超过总Task数的75%(可通过参数spark.speculation.quantile设置)时，再统计所有成功的Tasks的运行时间，得到一个中位数，用这个中位数乘以1.5(可通过参数spark.speculation.multiplier控制)得到运行时间门限，如果在运行的Tasks的运行时间超过这个门限，则对它启用推测。简单来说就是对那些拖慢整体进度的Tasks启用推测，以加速整个Stage的运行

spark.speculation.interval    100毫秒    Spark经常检查要推测的任务。

spark.speculation.multiplier    1.5    任务的速度比投机的中位数慢多少倍。

spark.speculation.quantile    0.75    在为特定阶段启用推测之前必须完成的任务的分数。

### SparkShuffle参数

#### spark.shuffle.consolidateFiles

默认值：false

参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。

调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。

#### spark.shuffle.file.buffer

默认值：32k

参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k，一定是成倍的增加），从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

#### spark.reducer.maxSizeInFlight

默认值：48m

参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

#### spark.shuffle.io.maxRetries

默认值：3

参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。

调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

taskScheduler不负责重试task，由DAGScheduler负责重试stage

#### spark.shuffle.io.retryWait

默认值：5s

参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。

调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

#### spark.shuffle.memoryFraction

默认值：0.2

参数说明：该参数代表了Executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%。

调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升10%左右。

#### spark.shuffle.manager

默认值：sort|hash

参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。

调优建议：由于SortShuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的SortShuffleManager就可以；而如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的HashShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

#### spark.shuffle.sort.bypassMergeThreshold

默认值：200

参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

调优建议：当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

#### spark.shuffle.consolidateFiles

默认值：false

参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。

调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。

### SparkStreaming Back Pressure参数

Spark Streaming 反压机制是1.5版本推出的特性，用来解决处理速度比摄入速度慢的情况，简单来讲就是做流量控制。

当批处理时间(Batch Processing Time)大于批次间隔(Batch Interval，即BatchDuration)时,说明处理数据的速度小于数据摄入的速度，持续时间过⻓或源头数据暴增，容易造成 数据在内存中堆积，最终导致Executor OOM。反压就是来解决这个问题的。  
 spark streaming的消费数据源方式有两种：  
 1. 若是基于Receiver的数据源，可以通过设置spark.streaming.receiver.maxRate来控制最大输入 速率；--Receiver模式已经淘汰  
 2. 若是基于Direct的数据源(如Kafka Direct Stream)，则可以通过设置spark.streaming.kafka.maxRatePerPartition来控制最大输入速率。

当然，在事先经过压测，且流量高峰不会超过预期的情况下，设置这些参数一般没什么问题。但最大值不代表是最优值，最好还能根据每个批次处理情况来动态预估下个批次最优速率。

在Spark 1.5.0以上，就可通过背压机制来实现。开启反压机制，即设置spark.streaming.backpressure.enabled为true，Spark Streaming会自动根据处理能力来调整输入速率，从而在流量高峰时仍能保证最大的吞吐和性能

* Spark Streaming的反压机制中，有以下几个重要的组件：

1:RateController  
 RateController 组件是 JobScheduler 的监听器，主要监听集群所有作业的提交、运行、完成情况，并从 BatchInfo 实例中获取以下信息，交给速率估算器（RateEstimator）做速率的估算:

-当前批次任务处理完成的时间戳 （processingEndTime）

-该批次从第一个 job 到最后一个 job 的实际处理时⻓ （processingDelay）

-该批次的调度时延，即从被提交到 JobScheduler 到第一个 job 开始处理的时⻓  
（schedulingDelay）

-该批次输入数据的总条数（numRecords）

2:RateEstimator  
 Spark 2.x 只支持基于 PID 的速率估算器，这里只讨论这种实现。基于 PID 的速率估算器简单地说就是它把收集到的数据（当前批次速率）和一个设定值（上一批次速率）进行比较，然后用它们 之间的差计算新的输入值，估算出一个合适的用于下一批次的流量阈值。这里估算出来的值就是流 量的阈值，用于更新每秒能够处理的最大记录数

3:RateLimiter

RateController和RateEstimator组件都是在Driver端用于更新最大速度的，而RateLimiter是用于接收到Driver的更新通知之后更新Executor的最大处理速率的组件。RateLimiter是一个抽象类，它并不是Spark本身实现 的，而是借助了第三方Google的GuavaRateLimiter来产生的。它实质上是一个限流器，也可以叫做令牌，如果Executor中task每秒计算的速度大于该值则阻塞，如果小于该值则通过，将流数据加入缓存中进行计算。

* 注意

\*反压机制真正起作用时需要至少处理一个批：由于反压机制需要根据当前批的速率，预估新批的速率，所以反压，机制真正起作用前，应至少保证处理一个批。

\*如何保证反压机制真正起作用前应用不会崩溃：要保证反压机制真正起作用前应用不会崩溃,需要控制每个批次

最大摄入速率。若为Direct Stream，如Kafka Direct Stream,则可以通过

spark.streaming.kafka.maxRatePerPartition参数来控制。此参数代表了每秒每个分区最大摄入的数据

条数。假设BatchDuration为10秒,spark.streaming.kafka.maxRatePerPartition为12条,

kafka topic分区数为3个，则一个批(Batch)最大读取的数据条数为360条(3\*12\*10=360)。同时，需要

注意，该参数也代表了整个应用生命周期中的最大速率，即使是背压调整的最大值也不会超过该参数。

* 使用SparkStreaming时有几个比较重要的参数：

|  |  |  |
| --- | --- | --- |
| 参数名称 | 默认值 | 说明 |
| spark.streaming.stopGracefullyOnShutdown | fasle | 设置为true确保在kill任务时，能够处理完最后一批数据，再关闭程序，不会发生强制kill导致数据处理中断，没处理完的数据丢失 |
| spark.streaming.backpressure.enabled | false | 是否启用反压机制，开启后spark自动根据系统负载选择最优消费速率，在 spark.streaming.backpressure.enabled开启的情况下，设置为整数，限制第一次批处理应该消费的数据，因为程序冷启动队列里面有大量积压，防止第一次全部读取，造成系统阻塞 |
| spark.streaming.kafka.maxRatePerPartition | 无 | 限制每秒每个消费线程读取每个kafka分区最大的数据量  默认直接读取所有，可以设置为整数  举例说明： |
| spark.streaming.backpressure.initialRate | 无 | 初始最大接收速率。只适用于Receiver Stream，不适用于Direct Stream。 |
| spark.streaming.backpressure.rateEstimator | pid | 速率控制器,Spark 默认只支持此控制器，可自定义，但没必要，用默认值即可。 |
| spark.streaming.backpressure.pid.proportional | 1.0 | 只能为非负值。当前速率与最后一批速率之间的差值对总控制信号贡献的权重。用默认值即可。 |
| spark.streaming.backpressure.pid.integral | 0.2 | 只能为非负值。比例误差累积对总控制信号贡献的权重。用默认值即可 |
| spark.streaming.backpressure.pid.derived | 0 | 只能为非负值。比例误差变化对总控制信号贡献的权重。用默认值即可 |
| spark.streaming.backpressure.pid.minRate | 100 | 只能为正数，最小速率 |

注意：

spark.streaming.kafka.maxRatePerPartition

限制每秒每个消费线程读取每个kafka分区最大的数据量

默认直接读取所有，可以设置为整数

举例说明：

BatchInterval：5s、Topic-Partition：3、maxRatePerPartition： 10000

最大消费数据量：10000 \* 3 \* 5 = 150000 条

### Spark3.0参数调优

<https://www.iteblog.com/archives/9839.html>

动态分区裁剪需要将

spark.sql.optimizer.dynamicPartitionPruning.enabled 参数设置为 true（默认为 true）

其他相关参数：

1.spark.sql.optimizer.dynamicPartitionPruning.useStats：默认为 true

When true, distinct count statistics will be used for computing the data size of the partitioned table after dynamic partition pruning, in order to evaluate if it is worth adding an extra subquery as the pruning filter if broadcast reuse is not applicable.

2.spark.sql.optimizer.dynamicPartitionPruning.fallbackFilterRatio：默认0.5

When statistics are not available or configured not to be used, this config will be used as the fallback filter ratio for computing the data size of the partitioned table after dynamic partition pruning, in order to evaluate if it is worth adding an extra subquery as the pruning filter if broadcast reuse is not applicable.

3.spark.sql.optimizer.dynamicPartitionPruning.reuseBroadcast：默认为 true

When true, dynamic partition pruning will seek to reuse the broadcast results from a broadcast hash join operation.

### 资源参数参考示例

<http://spark.apache.org/docs/latest/tuning.html>

以下是一份spark-submit命令的示例，大家可以参考一下，并根据自己的实际情况进行调节：

./bin/spark-submit \

--master yarn-cluster \

--num-executors 100 \

--executor-memory 6G \

--executor-cores 4 \

--driver-memory 1G \

--conf spark.default.parallelism=1000 \

--conf spark.storage.memoryFraction=0.5 \

--conf spark.shuffle.memoryFraction=0.3 \

*Run application locally on 8 cores*

./bin/spark-submit **\**

**--class** org.apache.spark.examples.SparkPi **\**

**--master** local[8] **\**

/path/to/examples.jar **\**

100

*# Run on a Spark standalone cluster in client deploy mode*

./bin/spark-submit **\**

**--class** org.apache.spark.examples.SparkPi **\**

**--master** spark://207.184.161.138:7077 **\**

**--executor-memory** 20G **\**

**--total-executor-cores** 100 **\**

/path/to/examples.jar **\**

1000

*# Run on a Spark standalone cluster in cluster deploy mode with supervise*

./bin/spark-submit **\**

**--class** org.apache.spark.examples.SparkPi **\**

**--master** spark://207.184.161.138:7077 **\**

**--deploy-mode** cluster **\**

**--supervise** **\**

**--executor-memory** 20G **\**

**--total-executor-cores** 100 **\**

/path/to/examples.jar **\**

1000

*# Run on a YARN cluster*export HADOOP\_CONF\_DIR=XXX

./bin/spark-submit **\**

**--class** org.apache.spark.examples.SparkPi **\**

**--master** yarn **\**

**--deploy-mode** cluster **\**  *# can be client for client mode*

**--executor-memory** 20G **\**

**--num-executors** 50 **\**

/path/to/examples.jar **\**

1000

*# Run a Python application on a Spark standalone cluster*

./bin/spark-submit **\**

**--master** spark://207.184.161.138:7077 **\**

examples/src/main/python/pi.py **\**

1000

*# Run on a Mesos cluster in cluster deploy mode with supervise*

./bin/spark-submit **\**

**--class** org.apache.spark.examples.SparkPi **\**

**--master** mesos://207.184.161.138:7077 **\**

**--deploy-mode** cluster **\**

**--supervise** **\**

**--executor-memory** 20G **\**

**--total-executor-cores** 100 **\**

http://path/to/examples.jar **\**

1000

*# Run on a Kubernetes cluster in cluster deploy mode*

./bin/spark-submit **\**

**--class** org.apache.spark.examples.SparkPi **\**

**--master** k8s://xx.yy.zz.ww:443 **\**

**--deploy-mode** cluster **\**

**--executor-memory** 20G **\**

**--num-executors** 50 **\**

http://path/to/examples.jar **\**

1000

注意：

根据实践经验来看，大部分Spark作业经过本次基础篇所讲解的开发调优与资源调优之后，一般都能以较高的性能运行了，足以满足我们的需求。但是在不同的生产环境和项目背景下，可能会遇到其他更加棘手的问题（比如各种数据倾斜），也可能会遇到更高的性能要求。为了应对这些挑战，需要使用更高级的技巧来处理这类问题。

资源参数的调优，没有一个固定的值，需要同学们根据自己的实际情况（包括Spark作业中的shuffle操作数量、RDD持久化操作数量以及spark web ui中显示的作业gc情况），同时参考课件中给出的原理以及调优建议，合理地设置上述参数。

## 避免数据倾斜

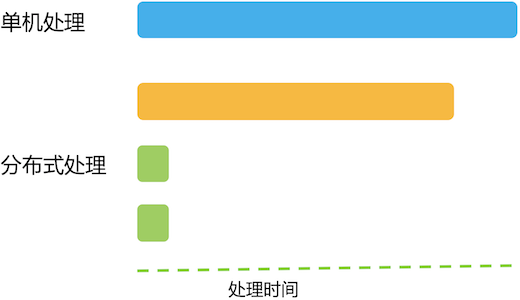
对于数据倾斜，并无一个统一的一劳永逸的方法。更多的时候，是结合数据特点（数据集大小，倾斜Key的多少等）综合使用多种方法。

### 什么是数据倾斜

对Spark/Hadoop这样的大数据系统来讲，数据量大并不可怕，可怕的是数据倾斜。

何谓数据倾斜？数据倾斜指的是，并行处理的数据集中，某一部分（如Spark或Kafka的一个Partition）的数据显著多于其它部分，从而使得该部分的处理速度成为整个数据集处理的瓶颈。

对于分布式系统而言，理想情况下，随着系统规模（节点数量）的增加，应用整体耗时线性下降。如果一台机器处理一批大量数据需要120分钟，当机器数量增加到三时，理想的耗时为120 / 3 = 40分钟，如下图所示

  
　　  
 但是，上述情况只是理想情况，实际上将单机任务转换成分布式任务后，会有overhead开销，使得总的任务量较之单机时有所增加，所以每台机器的执行时间加起来比单台机器时更大。这里暂不考虑这些overhead，假设单机任务转换成分布式任务后，总任务量不变。  
　　  
 但即使如此，想做到分布式情况下每台机器执行时间是单机时的1 / N，就必须保证每台机器的任务量相等。不幸的是，很多时候，任务的分配是不均匀的，甚至不均匀到大部分任务被分配到个别机器上，其它大部分机器所分配的任务量只占总得的小部分。比如一台机器负责处理80%的任务，另外两台机器各处理10%的任务，如下图所示  
  
　　  
在上图中，机器数据增加为三倍，但执行时间只降为原来的80%，远低于理想值。

### 数据倾斜的危害

从上图可见，当出现数据倾斜时，小量任务耗时远高于其它任务，从而使得整体耗时过大，未能充分发挥分布式系统的并行计算优势。  
　　  
 另外，当发生数据倾斜时，部分任务处理的数据量过大，可能造成内存不足使得任务失败，并进而引进整个应用失败。

### 数据倾斜是如何造成的

在Spark中，同一个Stage的不同Partition可以并行处理，而具有依赖关系的不同Stage之间是串行处理的。

假设某个Spark Job分为Stage 0和Stage 1两个Stage，且Stage 1依赖于Stage 0，那Stage 0完全处理结束之前不会处理Stage 1。而Stage 0可能包含N个Task，这N个Task可以并行进行。如果其中N-1个Task都在10秒内完成，而另外一个Task却耗时1分钟，那该Stage的总时间至少为1分钟。换句话说，一个Stage所耗费的时间，主要由最慢的那个Task决定。

由于同一个Stage内的所有Task执行相同的计算，在排除不同计算节点计算能力差异的前提下，不同Task之间耗时的差异主要由该Task所处理的数据量决定。

* Stage的数据来源主要分为如下两类

1.从数据源直接读取。如读取HDFS，Kafka

2.读取上一个Stage的Shuffle数据

### 避免数据源的数据倾斜-读Kafka

一般由于kafka主题的分区不均衡，一般由于key的hash不均匀，那么可以自定义分区器

如果没有key，可能是指定了分区造成的

以Spark Stream通过DirectStream方式读取Kafka数据为例。由于Kafka的每一个Partition对应Spark的一个Task（Partition），所以Kafka内相关Topic的各Partition之间数据是否平衡，直接决定Spark处理该数据时是否会产生数据倾斜。

Kafka某一Topic内消息在不同Partition之间的分布，主要由Producer端所使用的Partition实现类决定。如果使用随机Partitioner，则每条消息会随机发送到一个Partition中，从而从概率上来讲，各Partition间的数据会达到平衡。此时源Stage（直接读取Kafka数据的Stage）不会产生数据倾斜。

但很多时候，业务场景可能会要求将具备同一特征的数据顺序消费，此时就需要将具有相同特征的数据放于同一个Partition中。一个典型的场景是，需要将同一个用户相关的PV信息置于同一个Partition中。此时，如果产生了数据倾斜，则需要通过后面的其它方式处理。

### 避免数据源的数据倾斜-读文件

自定义分区的数量，spark.sql.shuffle.partitions,增大分区数

#### 原理

Spark以通过textFile(path, minPartitions)方法读取文件时，使用TextFileFormat。

对于不可切分的文件，每个文件对应一个Split从而对应一个Partition。此时各文件大小是否一致，很大程度上决定了是否存在数据源侧的数据倾斜。另外，对于不可切分的压缩文件，即使压缩后的文件大小一致，它所包含的实际数据量也可能差别很多，因为源文件数据重复度越高，压缩比越高。反过来，即使压缩文件大小接近，但由于压缩比可能差距很大，所需处理的数据量差距也可能很大。

此时可通过在数据生成端将不可切分文件存储为可切分文件，或者保证各文件包含数据量相同的方式避免数据倾斜。

对于可切分的文件，每个Split大小由如下算法决定。

其中goalSize等于所有文件总大小除以minPartitions。

而blockSize，如果是HDFS文件，由文件本身的block大小决定；如果是Linux本地文件，且使用本地模式，由fs.local.block.size决定。

protected long computeSplitSize(long goalSize, long minSize, long blockSize){

return Math.max(minSize, Math.min(goalSize, blockSize));

}

默认情况下各Split的大小不会太大，一般相当于一个Block大小（在Hadoop 2中，默认值为128MB），所以数据倾斜问题不明显。如果出现了严重的数据倾斜，可通过上述参数调整。

#### 总结

* 适用场景  
  数据源侧存在不可切分文件，且文件内包含的数据量相差较大。
* 解决方案  
  尽量使用可切分的格式代替不可切分的格式，或者保证各文件实际包含数据量大致相同。
* 优势  
  可撤底消除数据源侧数据倾斜，效果显著。
* 劣势  
  数据源一般来源于外部系统，需要外部系统的支持。

### 调整并行度分散同一个Task的不同Key

#### 原理

Spark在做Shuffle时，默认使用HashPartitioner（非Hash Shuffle）对数据进行分区。如果并行度设置的不合适，可能造成大量不相同的Key对应的数据被分配到了同一个Task上，造成该Task所处理的数据远大于其它Task，从而造成数据倾斜。

#### 总结

* 适用场景  
  大量不同的Key被分配到了相同的Task造成该Task数据量过大。
* 解决方案  
  调整并行度。一般是增大并行度，但有时减小并行度也可达到效果。
* 优势  
  实现简单，可在需要Shuffle的操作算子上直接设置并行度或者使用spark.default.parallelism设置。如果是Spark SQL，还可通过SET spark.sql.shuffle.partitions=[num\_tasks]设置并行度。可用最小的代价解决问题。一般如果出现数据倾斜，都可以通过这种方法先试验几次，如果问题未解决，再尝试其它方法。
* 劣势  
  适用场景少，只能将分配到同一Task的不同Key分散开，但对于同一Key倾斜严重的情况该方法并不适用。并且该方法一般只能缓解数据倾斜，没有彻底消除问题。从实践经验来看，其效果一般。

### 自定义Partitioner

#### 原理

使用自定义的Partitioner（默认为HashPartitioner），将原本被分配到同一个Task的不同Key分配到不同Task。

#### 总结

* 适用场景  
  大量不同的Key被分配到了相同的Task造成该Task数据量过大。
* 解决方案  
  使用自定义的Partitioner实现类代替默认的HashPartitioner，尽量将所有不同的Key均匀分配到不同的Task中。
* 优势  
  不影响原有的并行度设计。如果改变并行度，后续Stage的并行度也会默认改变，可能会影响后续Stage。
* 劣势  
  适用场景有限，只能将不同Key分散开，对于同一Key对应数据集非常大的场景不适用。效果与调整并行度类似，只能缓解数据倾斜而不能完全消除数据倾斜。而且需要根据数据特点自定义专用的Partitioner，不够灵活。

### 将Reduce side Join转变为Map side Join

#### 原理

通过Spark的Broadcast机制，将Reduce侧Join转化为Map侧Join，避免Shuffle从而完全消除Shuffle带来的数据倾斜。

#### 总结

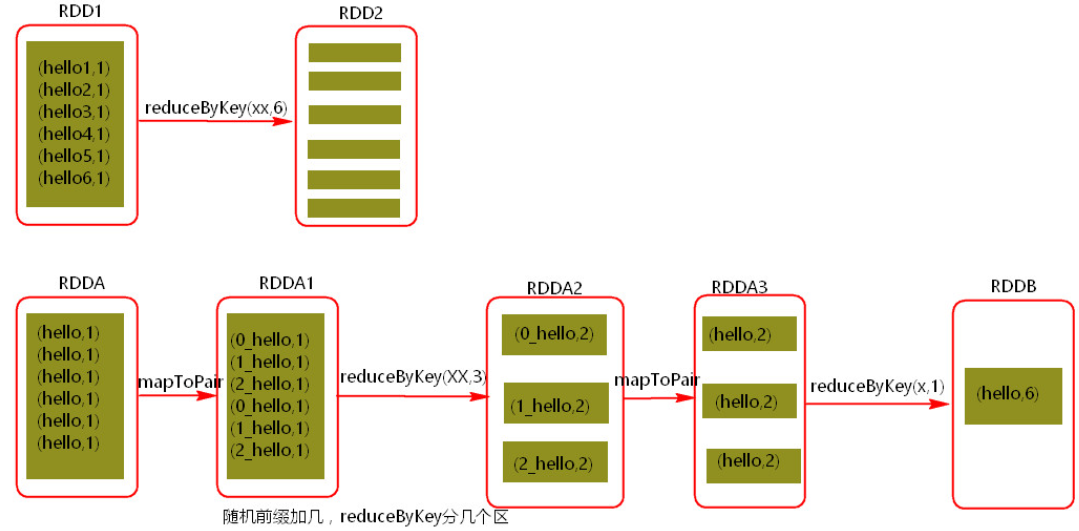
* 适用场景  
  参与Join的一边数据集足够小，可被加载进Driver并通过Broadcast方法广播到各个Executor中。
* 解决方案  
  在Java/Scala代码中将小数据集数据拉取到Driver，然后通过Broadcast方案将小数据集的数据广播到各Executor。或者在使用SQL前，将Broadcast的阈值调整得足够大，从而使用Broadcast生效。进而将Reduce侧Join替换为Map侧Join。
* 当数据集的大小小于spark.sql.autoBroadcastJoinThreshold 所设置的阈值的时候，使用广播join来代替hash join来优化join查询。广播join可以非常有效地用于具有相对较小的表和大型表之间的连接，它可以避免通过网络发送大表的所有数据
* 优势  
  避免了Shuffle，彻底消除了数据倾斜产生的条件，可极大提升性能。
* 劣势  
  要求参与Join的一侧数据集足够小，并且主要适用于Join的场景，不适合聚合的场景，适用条件有限。

### 为skew倾斜的key增加随机前/后缀

#### 原理

为数据量特别大的Key增加随机前/后缀，使得原来Key相同的数据变为Key不相同的数据，从而使倾斜的数据集分散到不同的Task中，彻底解决数据倾斜问题。

首先，通过map算子给每个数据的key添加随机数前缀，对key进行打散，将原先一样的key变成不一样的key，然后进行第一次聚合，这样就可以让原本被一个task处理的数据分散到多个task上去做局部聚合；随后，去除掉每个key的前缀，再次进行聚合。此方法对于由groupByKey、reduceByKey这类算子造成的数据倾斜由比较好的效果，仅仅适用于聚合类的shuffle操作，适用范围相对较窄。双重聚合:



如果是join类的shuffle操作可以将Join另一则的数据中，与倾斜Key对应的部分数据，与随机前缀集作笛卡尔乘积，从而保证无论数据倾斜侧倾斜Key如何加前缀，都能与之正常Join。如:

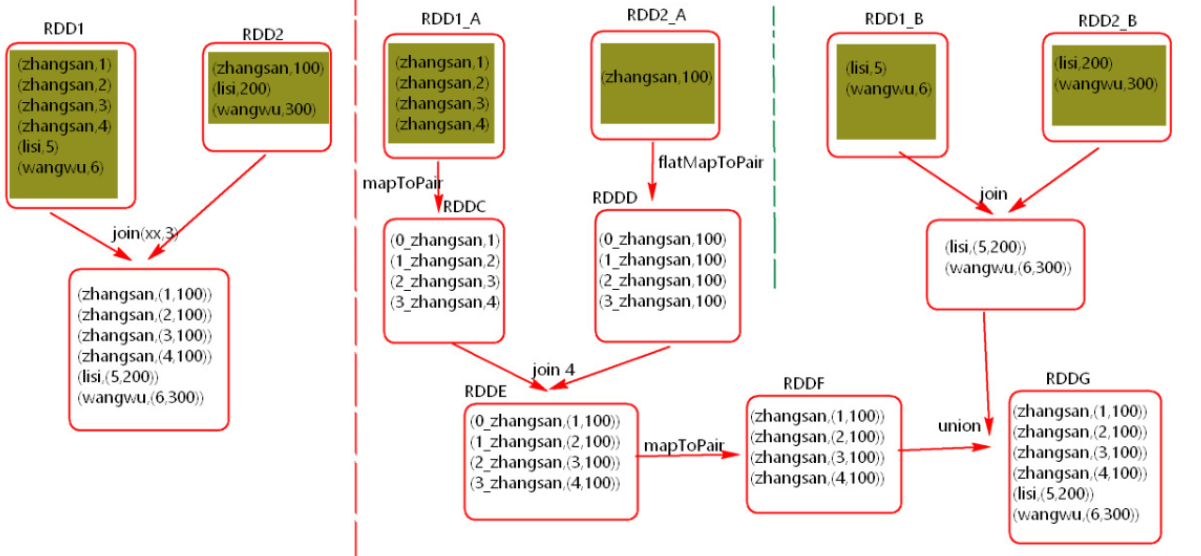
将leftRDD中倾斜的key对应的数据单独过滤出来，且加上1到n的随机前缀，并将前缀与原数据用逗号分隔（以方便之后去掉前缀）形成单独的leftSkewRDD

将rightRDD中倾斜key对应的数据抽取出来，并通过flatMap操作将该数据集中每条数据均转换为n条数据（每条分别加上1到n的随机前缀），形成单独的rightSkewRDD

将leftSkewRDD与rightSkewRDD进行Join，并将并行度设置为n，且在Join过程中将随机前缀去掉，得到倾斜数据集的Join结果skewedJoinRDD

将leftRDD中不包含倾斜Key的数据抽取出来作为单独的leftUnSkewRDD

对leftUnSkewRDD与原始的rightRDD进行Join，并行度也设置为n，得到Join结果unskewedJoinRDD

通过union算子将skewedJoinRDD与unskewedJoinRDD进行合并，从而得到完整的Join结果集  


#### 总结

* 适用场景  
  两张表都比较大，无法使用Map则Join。其中一个RDD有少数几个Key的数据量过大，另外一个RDD的Key分布较为均匀。
* 解决方案  
  将有数据倾斜的RDD中倾斜Key对应的数据集单独抽取出来加上随机前缀，另外一个RDD每条数据分别与随机前缀结合形成新的RDD（相当于将其数据增到到原来的N倍，N即为随机前缀的总个数），然后将二者Join并去掉前缀。然后将不包含倾斜Key的剩余数据进行Join。最后将两次Join的结果集通过union合并，即可得到全部Join结果。
* 优势  
  相对于Map则Join，更能适应大数据集的Join。如果资源充足，倾斜部分数据集与非倾斜部分数据集可并行进行，效率提升明显。且只针对倾斜部分的数据做数据扩展，增加的资源消耗有限。
* 劣势  
  如果倾斜Key非常多，则另一侧数据膨胀非常大，此方案不适用。而且此时对倾斜Key与非倾斜Key分开处理，需要扫描数据集两遍，增加了开销。

### 大表随机添加N种随机前缀，小表扩大N倍

#### 原理

如果出现数据倾斜的Key比较多，上一种方法将这些大量的倾斜Key分拆出来，意义不大。此时更适合直接对存在数据倾斜的数据集全部加上随机前缀，然后对另外一个不存在严重数据倾斜的数据集整体与随机前缀集作笛卡尔乘积（即将数据量扩大N倍）。

#### 总结

* 适用场景  
  一个数据集存在的倾斜Key比较多，另外一个数据集数据分布比较均匀。
* 优势  
  对大部分场景都适用，效果不错。
* 劣势  
  需要将一个数据集整体扩大N倍，会增加资源消耗。

## Shuffle优化

### Spark Shuffle

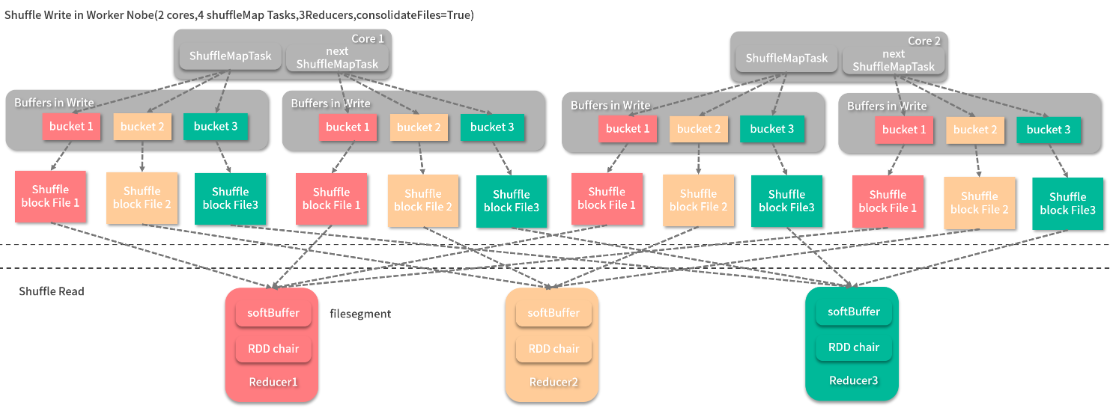
很多算子都会引起 RDD 中的数据进行重分区，新的分区被创建，旧的分区被合并或者被打碎，在重分区的过程中，如果数据发生了跨节点移动，就被称为 Shuffle，

在 Spark 中， Shuffle 负责将 Map 端（这里的 Map 端可以理解为宽依赖的左侧）的处理的中间结果传输到 Reduce 端供 Reduce 端聚合（这里的 Reduce 端可以理解为宽依赖的右侧），它是 MapReduce 类型计算框架中最重要的概念，同时也是很消耗性能的步骤。 Spark 对 Shuffle 的实现方式有两种：Hash Shuffle 与 Sort-based Shuffle，这其实是一个优化的过程。在较老的版本中，Spark Shuffle 的方式可以通过 spark.shuffle.manager 配置项进行配置，而在最新的 Spark 版本中，已经去掉了该配置，统一称为 Sort-based Shuffle。



### Hash Shuffle

在 Spark 1.6.3 之前， Hash Shuffle 都是 Spark Shuffle 的解决方案之一。 Shuffle 的过程一般分为两个部分：Shuffle Write 和 Shuffle Fetch，前者是 Map 任务划分分区、输出中间结果，而后者则是 Reduce 任务获取到的这些中间结果。Hash Shuffle 的过程如下图所示：



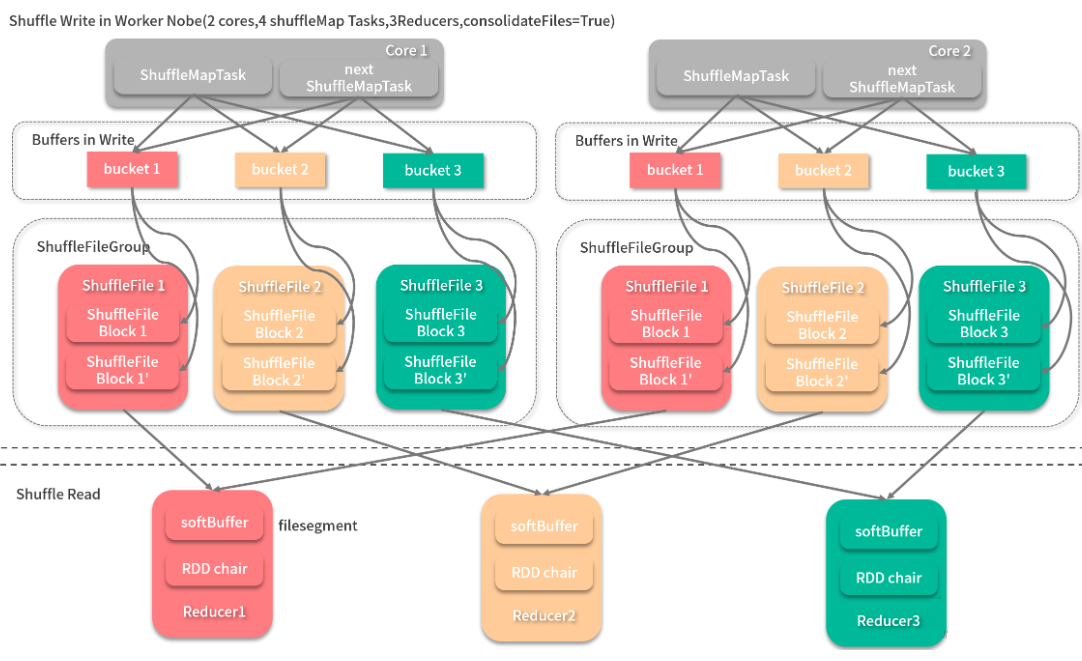
在图中，Shuffle Write 发生在一个节点上，该节点用来执行 Shuffle 任务的 CPU 核数为 2，每个核可以同时执行两个任务，每个任务输出的分区数与 Reducer（这里的 Reducer 指的是 Reduce 端的 Executor）数相同，即为 3，每个分区都有一个缓冲区（bucket）用来接收结果，每个缓冲区的大小由配置 spark.shuffle.file.buffer.kb 决定。这样每个缓冲区写满后，就会输出到一个文件段（filesegment），而 Reducer 就会去相应的节点拉取文件。这样的实现很简单，但是问题也很明显。主要有两个：

1.生成的中间结果文件数太大。理论上，每个 Shuffle 任务输出会产生 R 个文件（ R为Reducer 的个数），而 Shuffle 任务的个数往往由 Map 任务个数 M 决定，所以总共会生成 M \* R 个中间结果文件，而往往在一个作业中 M 和 R 都是很大的数字，在大型作业中，经常会出现文件句柄数突破操作系统限制。

2.缓冲区占用内存空间过大。单节点在执行 Shuffle 任务时缓存区大小消耗为 m \* R \* spark.shuffle.file.buffer.kb，m 为该节点运行的 Shuffle 任务数，如果一个核可以执行一个任务，m 就与 CPU 核数相等。这对于动辄有 32、64 物理核的服务器来说，是比不小的内存开销。

* File Consolidation

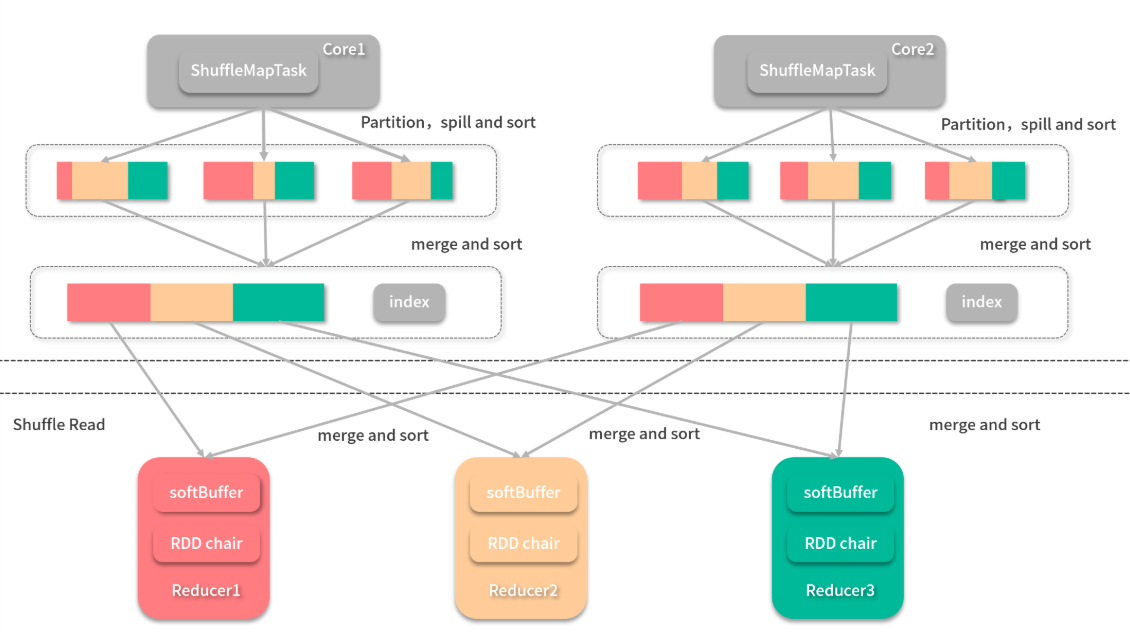
为了解决第一个问题， Spark 推出过 File Consolidation 机制，旨在通过共用输出文件以降低文件数，如下图所示：



每当 Shuffle 任务输出时，同一个 CPU 核心处理的 Map 任务的中间结果会输出到同分区的一个文件中，然后 Reducer 只需一次性将整个文件拿到即可。这样，Shuffle 产生的文件数为 C（CPU 核数）\* R。 Spark 的 FileConsolidation 机制默认开启，可以通过 spark.shuffle.consolidateFiles 配置项进行配置。

### Sort-based Shuffle

在 Spark 先后引入了 Hash Shuffle 与 FileConsolidation 后，还是无法根本解决中间文件数太大的问题，所以 Spark 在 1.2 之后又推出了与 MapReduce 一样的 Shuffle 机制： Sort-based Shuffle，才真正解决了 Shuffle 的问题，再加上 Tungsten 计划的优化， Spark 的 Sort-based Shuffle 比 MapReduce 的 Sort-based Shuffle 青出于蓝。如下图所示：



每个 Map 任务会最后只会输出两个文件（其中一个是索引文件），其中间过程采用的是与 MapReduce 一样的归并排序，但是会用索引文件记录每个分区的偏移量，输出完成后，Reducer 会根据索引文件得到属于自己的分区，在这种情况下，Shuffle 产生的中间结果文件数为 2 \* M（M 为 Map 任务数）。

* Bypass Sort-based

在基于排序的 Shuffle 中， Spark 还提供了一种折中方案——Bypass Sort-based Shuffle，当 Reduce 任务小于 spark.shuffle.sort.bypassMergeThreshold 配置（默认 200）时，Spark Shuffle 开始按照 Hash Shuffle 的方式处理数据，而不用进行归并排序，只是在 Shuffle Write 步骤的最后，将其合并为 1 个文件，并生成索引文件。这样实际上还是会生成大量的中间文件，只是最后合并为 1 个文件并省去排序所带来的开销，该方案的准确说法是 Hash Shuffle 的Shuffle Fetch 优化版。

Spark 在1.5 版本时开始了 Tungsten 计划，也在 1.5.0、 1.5.1、 1.5.2 的时候推出了一种 tungsten-sort 的选项，这是一种成果应用，类似于一种实验，该类型 Shuffle 本质上还是基于排序的 Shuffle，只是用 UnsafeShuffleWriter 进行 Map 任务输出，并采用了要在后面介绍的 BytesToBytesMap 相似的数据结构，把对数据的排序转化为对指针数组的排序，能够基于二进制数据进行操作，对 GC 有了很大提升。但是该方案对数据量有一些限制，随着 Tungsten 计划的逐渐成熟，该方案在 1.6 就消失不见了。

Tungsten号称Spark有史以来最大的改动，其致力于提升Spark程序对内存和CPU的利用率，使性能达到硬件的极限，主要工作包含以下三个方面

* Memory Management and Binary Processing 内存管理和二进制处理，降低对象的开销和消除JVM GC带来的延时。

　　1、避免使用非transient的Java对象（它们以二进制格式存储），这样可以减少GC的开销。

　　2、通过使用基于内存的密集数据格式，这样可以减少内存的使用情况。

　　3、更好的内存计算（字节的大小），而不是依赖启发式。

　　4、对于知道数据类型的操作（比如DataFrame和SQL），我们可以直接对二进制格式进行操作，这样我们就不需要进行系列化和反系列化的操作。

* Cache-aware Computation缓存感知计算.，优化存储，提升CPU L1/ L2/L3缓存命中率。

　　对aggregations, joins和shuffle操作进行快速排序和hash操作。

* Code Generation代码生成，优化Spark SQL的代码生成部分，提升CPU利用率

　　1、更快的表达式求值和DataFrame/SQL操作。

　　2、快速序列化

<https://www.csdn.net/article/2015-04-30/2824591-project-tungsten-bringing-spark-closer-to-bare-metal>

<https://blog.csdn.net/sundujing/article/details/51424491>

<https://www.jianshu.com/p/1d714f0c5e07>

## 优化案例

MapReduce作业大规模迁移Apache Spark在百度的实践

<https://blog.csdn.net/b6ecl1k7BS8O/article/details/84038957>

Spark SQL在100TB上的自适应执行实践

<https://www.cnblogs.com/shishanyuan/p/8453644.html>

字节跳动在 Spark SQL 上的核心优化实践<https://blog.csdn.net/csdnnews/article/details/103360235>